An adaptive real-time “uncommon-speed” alarm
Architecture documentation

Mark Triggs (3359647)
1 Functional description

1.1 Overview

Conceptually, the system consists of two main tasks: data recording and data analysis. The data recording part of the system takes data from the laser range finder and records the paths taken by objects that are observed. The data analysis part of the system inspects these recorded paths and sounds the alarm where appropriate.

The role of data recorder is fulfilled by the object tracker. This takes data from the laser range finder and compares it to a running average reading to find objects—collections of points that appear to belong to the same physical entity. It then correlates the newly-found objects with objects that were observed in prior readings. By stringing together the coordinates of observed objects over successive readings, the object tracker forms a path which models the motion of a physical entity.

The path repository stores the path information produced by the object tracker. It stores the paths taken by objects that are currently being tracked (current paths), and objects that were tracked previously (historic paths).

The monitor is responsible for the data analysis role. When new data are written to the path repository, the monitor attempts to find one or more similar historic paths for each current path. If a match for a current path cannot be found, or if the object represented by the current path is travelling at a significantly different speed to any of the similar historic paths, the alarm is sounded.

Figure 1: System overview
1.2 Object tracker

The object tracker task runs in a cycle of taking a reading from the laser range finder, finding the objects in that reading, matching them with current paths, and updating the path repository.

Conceptually, an object is a set of polar coordinates that belong to the same entity (a car, a person, etc.). These are represented within the system by three polar coordinates: the object’s midpoint, and the points at the object’s two extremes. Finding objects in a reading occurs in two stages: finding readings that differ from normal and are likely to represent some object, and grouping those readings into individual objects.

To find readings that differ from normal, the object tracker maintains an average distance for each angle in the laser reading\(^1\). When a reading is received, the object tracker compares the distance at each angle of the reading to the average distance for that angle. If the distance of the reading differs significantly from average, that point is assumed to be a part of some object.

Once object readings have been found, the object tracker uses a correlation function (discussed in Section 1.6.1) to group them into objects. Two points in a reading are grouped into the same object if they are sufficiently close (where “sufficiently close” is an adjustable parameter measured in centimetres). Once this is done, the laser reading is added to the “average reading”, to aid in detecting future objects.

Having identified the objects present in the current reading, the object tracker attempts to find out if each object is the continuation of a current path—paths formed by objects that are currently being tracked. To do this, it uses multiple tasks to find the closest match between newly-identified objects and current paths.\(^2\)

Once the object tracker has matched the objects found in the current reading to existing current paths, it passes the information to the path repository, which updates its stored object paths.

1.3 Path repository

The path repository records the paths calculated by the object tracker, and makes information available to both the object tracker and the monitor. Internally, it

---

\(^1\)This average is initialised during the calibration phase, discussed in Section 1.5.

\(^2\)More information on the real-time aspects of this is available in Section 2.3.
keeps track of two types of paths: paths taken by objects that are currently being tracked (current paths), and paths that were taken by objects that have since stopped being tracked (historic paths).

The path repository takes input from the object tracker in the form of a mapping from newly-found objects to current paths. In this mapping, it looks for three cases: newly-found objects that are the continuation of a current path, newly-found objects that do not continue a current path, and current paths that were not continued by any newly-found object.

When a newly-found object is indicated to be the continuation of a current path, it is simply appended to that current path.

If a newly-found object is not the continuation of any current path, it is assumed to be an object that has not been observed previously. A new entry is added to the list of current paths so that the object will be tracked in future readings.

Current paths that were not updated by the last laser reading (that is, the most recent reading did not yield a continuation to that path) are moved from the list of current paths to the list of historic paths. This indicates that the object that was being tracked has since been lost from view, and will not be tracked in subsequent readings. Historic path information is used by the monitor to decide whether a current path is “unusual”.

The path repository provides the following facilities:

- a facility that allows the object tracker to update the current paths by supplying a mapping from newly-found objects to paths
- facilities for the object tracker and monitor tasks to read the next item from the current or historic path list
- a facility for marking a current path as having sounded the alarm
- some means for the monitor to indicate that a historic path was successfully correlated with a current path
- some way for the monitor to set the age at which historic paths are “expired” (these last two facilities relate to historic path expiring, which is discussed in Section 1.4.1).

To allow multiple readers to get the next item from the list they are reading from, each list element would be assigned some unique identifier by the path repository. When a reader requests the next item in a list, they supply the identifier of the last item they read, and the path repository finds and returns the next item.
1.4 Monitor

The monitor task analyses the data placed in the path repository by the object tracker, and is responsible for examining the paths and speeds of objects that are currently being tracked. When the speed or path taken by such an object is “unusual”, the monitor immediately sounds an alarm.

An object may be classified as behaving “unusually” for two reasons: because it is travelling a path that has not been travelled by any object previously, or because it is travelling a known path at a speed that is significantly different from the speed normally travelled along that path. Thus, the essence of the monitor’s activities is to compare current paths to historic paths, comparing both the course followed and average speeds of the objects they represent.

To determine whether an object is travelling a known path, the monitor uses a number of sub-tasks to correlate the object’s path with a path travelled by any object previously. A current path is deemed to be normal if it is able to be correlated with a historic path. This means that each historic path defines the “normal behaviour” for its surrounding area, and in this way, dynamic regions are formed. Figure 2 illustrates this: the points shown in black indicate a historic path, while the points shown in grey indicate current paths that would correlate closely enough with that path to be considered “usual”.

![Figure 2: Dynamic regions are formed by correlation with the paths taken by previously tracked objects.](image)

If the current path being analysed by the monitor cannot be correlated with any historic path, the tracked object is assumed to be travelling an unusual path, and the alarm is sounded.

If the current path being analysed can be correlated with one or more historic paths, the monitor concludes that the object is travelling a known path. It next
checks whether the current object is moving at approximately the same speed along that path as the previous objects it correlates with. To do this, it calculates the average speed of the current path being analysed, and compares it to the average speeds of each of the historic paths it correlated with. If the speed of the current path varies significantly from the speed of every similar historic path, the alarm is sounded. Possible algorithms for calculating average speeds are discussed in Section 1.6.2.

Once a current path has been deemed “unusual” and the alarm has been sounded, it is desirable to continue tracking the object but not to analyse it further. To achieve this, each current path contains a flag that indicates whether it has caused the alarm to be sounded. This flag is set by the monitor when sounding the alarm, and indicates that the path can be ignored in future analysis.

1.4.1 Expanding historic paths

If all that was required for an object’s path to be considered “usual” was for it to correlate with a historic path that had already been observed, the results would be unsatisfactory. For example, an event that an observer would consider as unusual could happen twice, and the alarm would not be sounded the second time. The system could avoid this problem by not recording unusual paths in the path repository, but this would inhibit its adaptability to a changed environment. Additionally, the system cannot continue storing paths indefinitely, and must eventually discard some of the data.

This problem is addressed by ageing historic paths stored in the path repository, and “expiring” them once they reach a certain age. A path’s “age” reflects how recently it has been correlated with a current path. For example, a path that represents the motion of traffic across a busy road will often be correlated with, and will remain young. Conversely, a path that was recorded when an unusual event occurred will very rarely be correlated with (hence, “unusual”), and will eventually age and be expired.

When an object is initially added to the historic object list, its age is set to zero. Each time the monitor correlates the path of a currently tracked object with a historic path, the path repository resets the age of that historic path back to zero, and increments the age of every other historic path. If the age of any historic path has reached a certain threshold, the path repository discards that historic path.

Although expiry is carried out by the path repository itself, the age at which a historic path is expired is controlled by the monitor task. The monitor initially sets this to some default value, but may decide to adjust this value under certain
conditions (discussed in Section 2.5).

1.5 System operation

The system relies heavily on the information it collects about its environment. The object tracker needs to maintain an average reading to identify objects in new readings, and the monitor needs access to a repository of historic information about object paths. Before the system is able to operate normally, these information stores must be filled.

To achieve this, the system begins by going through two calibration phases. The first phase involves only the object tracker, which operates in a continuous cycle of taking a reading from the laser range finder and adding it to its running average reading. During the second phase, the object tracker operates as normal but the monitor task is not active. This time is spent by the object tracker identifying and tracking objects through successive readings, and updating the path repository.

Once the system has enough information to identify objects and detect unusual behaviour, the monitor task is activated and begins analysing the path repository.

1.6 Algorithms employed

1.6.1 Correlation algorithms

Three different (but related) correlation functions are used by the system: a function to group individual points into objects, a function to associate an object in one reading with the same object in prior readings (to form paths), and a function to compare current paths with historic paths. These must be present for the system to operate, but the particular implementation is left open; any of these could be replaced with a more “intelligent” algorithm to improve performance.

All three of these functions primarily deal with the object data type. This could be defined as follows:

```plaintext
type Object is record
    Start_Point: Point;
    Mid_Point: Point;
    End_Point: Point;
end record;
```

where a “point” is simply a polar coordinate:
type Point is record
  Angle: Degree;
  Distance: Laser_Distance;
end record;

Another important data type is the “path”, which is a sequence of objects that were found over successive readings. This could be defined as:

type Path_Range is new Integer range 0 .. Max_Path_Length;
type Object_List is array (Path_Range) of Object;
type Speed is new Metrics.cm range 0 .. Max_Possible_Speed;

type Path is record
  Objects: Object_List;
  N: Path_Range; -- The most recently observed object in this path
  Average_Speed: Speed;
  Average_Speed_Known: Boolean := False;
end record;

Fundamentally, the system must be able to identify objects that are present in a reading from the laser range finder. I have observed good results by defining an object as a sequence of points at successive degrees whose distances do not vary by more than 50cm (obviously this parameter could be tuned for a particular environment). Some pseudo-code for this sort of scheme might look like:

procedure Find_Objects (Reading: in Laser_Status;
  Objects: out Object_List;
  Object_Count: out Integer) is
  Count: Integer;
begin
  for each point P in Reading loop
    if abs (P.Distance - Last_P.Distance) < 50 then
      -- <add P to the current object>
    else
      -- add the current object to Objects, increment Count,
      if Count < Max_Path_Length then
        -- <start a new object>
      else
        exit;
      end if;
    end if;
    Last_P := P;
  end loop;

  Object_Count := Count;
end Find_Objects;
Once the objects in the current reading have been identified, the object tracker must determine if they are a continuation of a current path, or the start of a new one. This requires a measurement of correlation between objects, which could be implemented as follows:

```plaintext
function Object_Correlation (O1: Object; O2: Object) return Correlation is
    begin
    return (Point_Distance (O1.Start_Point, O2.Start_Point) +
            Point_Distance (O1.Mid_Point, O2.Mid_Point) +
            Point_Distance (O1.End_Point, O2.End_Point));
end Object_Correlation;
```

Using this correlation function, we can define a function which shows how closely an object found in a reading correlates with an existing current path. We could say that an object is likely to be the \( n \)th reading in a path if it is sufficiently closely correlated to the \( n-1 \)th reading in that path. Using this technique:

```plaintext
function Object_Path_Correlation (O: Object; P: Path) return Correlation is
    begin
    return Object_Correlation (O, P.Objects (P.N));
end Object_Path_Correlation;
```

The monitor task must also be able to compare to paths to determine whether their course is similar. To do this, a correlation function is needed to indicate how closely two paths follow the same course. This is done by using a point-wise comparison, as follows:

```plaintext
function Do_Paths_Correlate (P1: Path; P2: Path) return Boolean is
    Longer: Path;
    Shorter: Path;
    Path_Start: Path_Range;
    C: Correlation;
    Best_Correlation: Correlation;
    begin
    if P1.N > P2.N then
        Longer := P1; Shorter := P2;
    else
        Longer := P2; Shorter := P1;
    end if;

    -- Find the point where the longer and shorter paths begin
    -- to match.
    Path_Start := Path_Range'First;
    Best_Correlation := Object_Correlation (Longer.Objects (Path_Start),
                                           Shorter.Objects (Path_Start));
    for I in Path_Range'First .. Longer.N loop
```
C := Object_Correlation (Longer.Objects (I),
       Shorter.Objects (Path_Range'First));

if C < Best_Correlation then
    Best_Correlation := C;
    Path_Start := I;
end if;
end loop;

-- Check if the two paths are travelling the same course
for I in Path_Range'First .. Shorter.N loop
    if (I + Path_Start) < Longer.N then
        C := Object_Correlation (Shorter.Objects (I),
                           Longer.Objects (I + Path_Start));

        if C > Allowable_Difference then
            return False;
        end if;
    else
        -- The shorter path cannot be entirely matched with the
        -- longer path.
        return False;
    end if;
end loop;

return True;
end Path_Correlation;

This method of comparing the two paths point-by-point over their course in chronological order means that the direction of the paths is also considered. If the course of two paths is identical but the directions in which the objects travelled that course differs (e.g. a car driving down the wrong side of a road), the two paths will still not correlate.

1.6.2 Speed measurement algorithms

The monitor must calculate the average speed travelled along a path in order to decide whether an object is moving too fast or too slow. All the information required to do this is contained in the path: each object making up the path was recorded approximately $\frac{1}{5}$ of a second after the object preceding it, and this information can be used to calculate the average speed travelled along the path.

procedure Calculate_Average_Speed (P : out Path) is
    Average_Speed: Speed;
Point_Speed: Speed;
begin
    if not P.Average_Speed_Known then
        for I in Path_Range'First .. (P.N - 1) loop
            Point_Speed := Point_Distance (P.Objects (I).Mid_Point,
                                             P.Objects (I + 1).Mid_Point) * 5;
            -- <Add Point_Speed to Average_Speed>
        end loop;
        P.Average_Speed := Average_Speed;
        P.Average_Speed_Known := True;
    end if;
end Calculate_Average_Speed;
2 Temporal description

2.1 Overview

The logical separation of data recording (via the object tracker) and data analysis (via the monitor) is largely due to the timing requirements of those parts of the system. This separation reflects the fact that the object tracker and monitor must satisfy the timing requirements of different external sources.

Because it is important that the system record all available data, the timing of the object tracker is dictated by the frequency of readings from the laser range finder. Thus, the object tracker must be able to identify objects and correlate them with current paths within approximately a fifth of a second.

The monitor’s timing requirements do not come from the laser range finder, but from the users of the system—the people who are notified of an unusual event by the alarm. Thus, the timing that the monitor must exhibit is determined by the perceptions of the system’s users. It is reasonable to expect that these requirements will not be as stringent as the fifth of a second required by the laser range finder. Obviously sounding an alarm minutes after an unusual event is unsatisfactory, but sounding an alarm within a second will often be sufficient.

2.2 Guiding principles

The system tries to adhere to a number of design principles. These were considered when deciding between possible design strategies.

Timing over accuracy. An inaccurate result on time is better than an accurate result too late.

Exploit concurrency. Use multiple tasks to spread workload across processors where appropriate.

Simplicity. Favour a simple solution over a more complicated one. A more complicated solution might give better results, but a simple solution often has a better chance of satisfying timing requirements.

Avoid dynamic behaviour where possible. Avoid using data structures or tasks that must be allocated or created at run-time. Static storage and a fixed number of tasks helps to make timing predictable.
2.3 Object tracker

The object tracker must perform its task as quickly as possible in order to keep up with the laser range finder. When it receives a reading from the laser range finder, it must identify the different objects in that reading, correlate each of those objects with existing current paths if possible, and update the path repository, all before the next reading becomes available.

Assuming a relatively simple algorithm (such as the one outlined in Section 1.6.1), finding objects in a reading can be made fast and quite predictable. This requires a single traversal of the laser reading data, and objects to be added to a running list of objects. Constraining the maximum number of objects that can be found in a reading also allows a static sized array to be used, improving the predictability of this process’s timing. In the worst case, the entire array will need to be scanned and the maximum number of objects will be found.

Correlating each object identified with a current path is inherently unpredictable, as the amount of processing required depends on how many objects are currently being tracked. This in turn depends on the system’s external environment.

For each object found in the laser reading, the object tracker must calculate how closely it correlates with each current path, forming a table as shown in Table 1.

<table>
<thead>
<tr>
<th>Current path 1</th>
<th>Current path 2</th>
<th>Current path 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object 1</td>
<td>40</td>
<td>900</td>
</tr>
<tr>
<td>Object 2</td>
<td>50</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 1: Correlation values calculated by the object tracker

To minimise the time spent calculating the values of this table, the object tracker makes use of worker tasks that can be spread across multiple processors, where each task creates a row of this table. These tasks are created when the system is started, and run in a loop of accepting an object and a list of current paths from the main object tracker task, calculating the correlations and then returning the results to the main object tracker task. The object tracker collects these results, calculates the closest matches between objects and current paths and updates the path repository.

The correlation process does as much work as it can until the next laser reading arrives. If the helper tasks are still calculating when the next laser reading arrives, the object tracker immediately instructs those tasks to abort, writes the partial results to the path repository and begins processing the next reading.
The relevant portion of the main object tracker task might look something like:

```pascal
select
  Laser.Get_Next_Reading (Sample)
Status.Stop; -- notify worker tasks
Aborted := True;
then abort
  Next_Worker := Workers'First;
  for each object O in Objects loop
    if Worker_Status (Next_Worker).Has_Results then
      Worker(I).Finish (Result);
      -- <Add Result to the table of results>
    end if;
    -- Give them some more work
    Worker (Next_Worker).Start (O, Current_Paths);
    Next_Worker := Next_Worker + 1; -- A mod range;
  end loop;
  -- Collect any remaining results
  for I in Worker'Range loop
    if not Aborted and (Worker_Status (I).Has_Results or
      Worker_Status (I).Calculating) then
      Worker (I).Finish (Result);
      -- <Add Result to the table of results>
    end if;
  end loop;
end select;
```

While the helper task might look like:

```pascal
loop
  accept Start (Obj : Object; Current_Paths : Path_List);
  -- Perform the correlation but abort when requested
  select
    Status.Aborted;
    Aborted := True;
  then abort
    Worker_Status (My_ID).Calculating := True;
    -- <Calculate the correlations for each Current_Path>
  end select;
  Worker_Status (My_ID).Calculating := False;
  if not Aborted then
```
-- We have results to give
Worker_Status (My_ID).Has_Results := True;
accept Finish (Results : out Mapping) do
  Results := Res;
  Worker_Status (My_ID).Has_Results := False;
end;
end if;
end loop;

Using this method, objects found in the current reading may not be correlated with anything if the correlation process runs out of time. The effect of this is not as bad as it sounds: such objects will be registered as starting a new current path when tracked through future readings.

2.4 Path repository

The path repository must service read and update requests from the object tracker and read requests from the monitor. Because of their differences in timing requirements, the object tracker should always be given priority over the monitor.

In a sense, the path repository is a form of the readers and writers problem. Multiple reads can happen concurrently (for example, both the monitor and object tracker can read from the list of current paths simultaneously), but when updating the list of current paths, the object tracker should be given exclusive access. If the object tracker wishes to update the list of current paths while the monitor wishes to read them, the update operation (the write) should always be given priority. This makes sense from a timing perspective, as the object tracker runs under more stringent timing requirements, but also from a functional perspective, as the monitor should analyse the most up-to-date information.

The path repository gives access to its stored information one path at a time, instead of providing the entire list of paths. Although this makes the interaction between the tasks and the path repository slightly more complex, it comes with two advantages: the amount of data copied around is minimised, as only a single path need be copied at at time, and the risk of a task storing the whole list and working off of outdated information is reduced.

2.5 Monitor

Like the object tracker, the monitor makes use of worker tasks spread across multiple processors to carry out its path checking quickly. The scheme used here is
similar to that of the object tracker: the monitor runs in a loop of allocating a path from the current path list to each worker task, and the worker tasks check if their path correlates with any historic path and either note the correlation in the path repository or sound the alarm. When the monitor has checked all the current paths, it begins the process again.

The monitor’s timing comes from what is perceived as a reasonable response time by the system’s users. Although this can be quite subjective, it seems reasonable to assume that the monitor should respond to an unusual event within one second of its occurrence—that is, the monitor should inspect each current path every second.

The most likely reason for the monitor not to be able to meet this deadline is if there are too many historic paths stored in the path repository. Increasing the number of historic paths can potentially increase the searching space that the monitor must check when correlating current paths.

To avoid not meeting deadlines, the monitor runs in two modes of operation. In the normal mode of operation, where the deadline is being met, the monitor behaves as described, allocating work to its subtasks who record their correlations in the path repository and possibly sound the alarm. The second mode of operation is triggered when the monitor fails to meet its deadline. When this occurs, the monitor immediately aborts its worker tasks and requests that the path repository set the new expiry age to half of its original value. This causes the path repository to discard elements from the historic path list and increases the likelihood of the monitor meeting its next deadline. After doing this, the monitor resumes its normal mode of operation, but also increments the expiry age by 10% each time the deadline is successfully met, until it eventually returns to its original value. By doing this, the monitor automatically regulates the amount of work it does based on its timing constraints.